Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
J. Parallel Distrib. Comput. 65 (2005) 667 – 677
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Short note

Modeling and optimization of non-blocking checkpointing for optimistic simulation on myrinet clusters☆

Francesco Quaglia*, Andrea Santoro

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy

Received 10 September 2003; accepted 15 February 2005
Available online 8 April 2005

Abstract

Checkpointing-and-Communication Library (CCL) is a recently developed software which implements CPU offloaded, non-blocking checkpointing functionalities in support of optimistic parallel simulation on myrinet clusters. This is achieved by exploiting data transfer capabilities provided by a programmable DMA engine on board of myrinet network cards. Re-synchronization between CPU and DMA activities must sometimes be employed for several reasons, such as the maintenance of data consistency, thus adding overhead to (otherwise CPU cost-free) non-blocking checkpoint operations. In this paper we present a detailed cost model for non-blocking checkpointing and derive a performance effective re-synchronization semantic which we call *minimum cost re-synchronization*. With this semantic, an occurrence of re-synchronization either commits an on-going DMA based checkpoint operation (causing suspension of CPU activities) or aborts the operation (with possible increase in the expected rollback cost due to a reduced amount of committed checkpoints) on the basis of a minimum overhead expectation evaluated through the cost model.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Parallel discrete-event simulation; Checkpointing; Optimistic synchronization; Rollback-recovery; Myrinet; DMA; COTS; Performance optimization

1. Introduction

A Checkpointing-and-Communication Library (CCL) providing CPU offloaded, non-blocking checkpointing functionalities has been recently implemented [17,22] in support of optimistic parallel simulation [6,7] on myrinet clusters, i.e., a Commercial Off-The-Shelf (COTS) architecture recognized as a standard for parallel computing applications [9]. In CCL a checkpoint operation, i.e., memory-to-memory transfer of the Logical Process (LP) state vector into the checkpoint buffer, is charged on a programmable DMA engine on board of myrinet network cards, thus allowing the CPU to carry out other simulation specific operations while checkpointing is in progress.

This solution has shown the potential for improving the execution speed of the parallel simulation application thanks to a reduction of the CPU overhead due to checkpointing [17,21]. However, an important performance issue to address is related to the fact that the CPU and the DMA engine must sometimes "re-synchronize" due to a set of reasons. Just to name one, any LP scheduled for event execution cannot proceed while a DMA based checkpoint operation involving its state vector is in progress, otherwise that checkpoint might result in an incorrect snapshot of the LP state.

Re-synchronization has been implemented within CCL according to a simple threshold based semantic, called *Conditional Checkpoint Abort* ($\mathcal{CCA}$) [17], which determines whether a checkpoint should be committed/aborted upon re-synchronization on the basis of the advancement percentage of the checkpoint operation carried out by the DMA. Although run-time tuning of the threshold percentage can be performed [17], $\mathcal{CCA}$ might still suffer from performance weakness since it does not rely on the evaluation of real

☆ An earlier version of this paper appeared in Proceedings of 17th ACM International Conference on Supercomputing, June 2003.

* Corresponding author. Fax: +39 06 85300849.
E-mail address: quaglia@dis.uniroma1.it (F. Quaglia).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.02.006

commit/abort costs (e.g., in terms of freezing of CPU activities when a checkpoint needs to be committed) possibly impacting the performance of the simulation system.

In this paper, we present a detailed cost model for non-blocking checkpointing, and derive a *Minimum Cost* ($\mathcal{MC}$) re-synchronization semantic based on the model, which takes the checkpoint commit/abort decision on the basis of a minimum overhead expectation. In order to solve the model, we need to estimate/determine the value of low level system parameters, such as the residual completion latency for DMA transfers associated with the on-going non-blocking checkpoint operation. While presenting software extensions to support $\mathcal{MC}$, we show how these parameters can be estimated/evaluated in the specific implementation of non-blocking checkpointing within CCL. We also report experimental results for a Personal Communication System (PCS) simulation application, selected as a testbed, which quantify the performance improvements achievable through $\mathcal{MC}$.

The remainder of this paper is organized as follows. Section 2 provides a detailed overview of CCL. The cost model for non-blocking checkpointing and the $\mathcal{MC}$ re-synchronization semantic, including implementation details, are presented in Section 3. Experimental results are reported in Section 4.

2. Overview of CCL

CCL [17,22] has been designed for computer clusters connected through myrinet networks [10]. A myrinet network card is a programmable communication device, based on the LANai chip [11,12], consisting of: (A) An internal bus, namely LBUS (Local BUS); (B) A programmable processor connected to the LBUS, which we will refer to as LANai processor; (C) A RAM bank (LANai internal memory), connected to the LBUS, which can be mapped into the memory address space of the host; (D) A packet interface between the myrinet switch and the LANai processor; (E) Three DMA engines used respectively for (i) packet-interface/internal-memory transfer (Receive DMA), (ii) internal-memory/packet-interface transfer (Send DMA), (iii) internal-memory/host-memory transfer or vice-versa (EBUS DMA, namely External Bus DMA). Host access to the LANai internal memory takes place through a PCI bridge, which is also used for EBUS DMA data transfer from the host memory to the LANai internal memory and vice versa. As in the common choice to fast speed messaging layers for myrinet (see for example [14]), in CCL messages incoming from the network are temporarily buffered into the LANai internal memory (data transfer between the packet interface and the internal memory takes place through the Receive DMA) and then transferred into the receive queue, located onto host memory, through the EBUS DMA. Following another common design choice, any send operation issued by the application involves copying the message content directly into the LANai internal memory. Then the message is transferred onto the network through the Send DMA.

Actually, the EBUS DMA is used not only to transfer messages from the LANai internal memory to the receive queue, but also for data transfer associated with checkpointing. Specifically, a checkpoint operation involves data transfer from the LP state buffer (located onto host memory) to the stack of the checkpointed state vectors of the LP (also located onto host memory). The transfer operation is charged on the EBUS DMA that uses the LANai internal memory as a temporary buffer.

The responsibility to program the three DMA engines anytime there is the need for supporting a given data transfer operation pertains to a *control program* run by the LANai processor. Therefore, issuing a checkpoint request at the application level actually means requesting the LANai processor to program the EBUS DMA for the data transfer operation. This is done by the function `non_block_ckpt(int LP_id, double simulation_clock)`, included in the API provided by CCL, where `LP_id` is the identifier of the LP whose state vector needs to be checkpointed, and `simulation_clock` is the value of the current simulation time of the LP.[1] Any checkpoint operation is split by the control program into a sequence of data transfer operations to be performed by the EBUS DMA. Each operation transfers up to a maximum amount of bytes, called *burst*, from the LP state vector to the LANai internal memory (intermediate buffering) or from the LANai internal memory to the stack of checkpoints of the LP. Also, lower priority is assigned to data transfer associated with checkpointing as compared to message transfer into the receive queue. These two engineering choices allow checkpointing functionalities to produce negligible interference with communication functionalities. Specifically, splitting any checkpoint operation into a sequence of bursts allows prompt re-assignment of the hardware resources on board of the myrinet card (i.e., the EBUS DMA, the PCI bridge and the LBUS) to communication operations due to their higher priority.[2]

Re-synchronization between CPU activities and checkpointing activities carried out by the EBUS DMA is required to avoid data inconsistency (i.e., updating an LP state vector that is still being transferred into the checkpoint buffer through the EBUS DMA) and to prevent contention on the hardware due to activation of multiple checkpoint requests. The re-synchronization functionality should be activated whenever the LP that lastly issued a checkpoint re-

[1] CCL manages the checkpoint stacks of the LPs in a totally transparent way to the application programmer [17], which is the reason why `LP_id` is a sufficient parameter to identify both the state buffer and the entry into the stack of checkpoints that must be involved in the data transfer.

[2] Ref. [18] shows how to identify the maximum value for the burst length, which allows the performance of communication functionalities not to be significantly perturbed by the activation of checkpointing functionalities. The use of such a maximum value is recommended since it keeps the checkpoint latency at a minimum by keeping low the amount of bursts required for the checkpoint operation.

quest is re-scheduled for event execution (this prevents data inconsistency) and also before issuing any new checkpoint request (this prevents hardware contention).

As pointed out in the Introduction, the re-synchronization functionality currently offered by CCL is based on the $\mathcal{CCA}$ semantic [17]. To implement $\mathcal{CCA}$, CCL maintains a counter in the LANai internal memory, namely `completed_transfers`, which is reset by the `non_block_ckpt()` function upon the issue of a checkpoint request at the application level, and is incremented by the control program each time the program becomes aware that an EBUS DMA data transfer (from/to host memory) associated with the checkpoint operation has been completed. The value of this counter is used by the re-synchronization function `ckpt_cond_abort(float threshold)`, where the parameter `threshold` indicates the completion percentage of the last activated checkpoint operation, if any, under which the checkpoint operation must be aborted. To decide whether to abort the checkpoint operation or not, `ckpt_cond_abort()` needs information about both the current value of the `completed_transfers` counter and the total number of EBUS DMA data transfers required for the operation. The latter information is maintained into an additional variable located onto host memory, namely `total_transfers`, which is visible to `ckpt_cond_abort()`, as well as to the function `non_block_ckpt()`. As soon as the checkpoint operation is issued by the application, the function `non_block_ckpt()` computes the total number of EBUS DMA transfers (from/to host memory) to complete the operation. The variable `total_transfers` stores the obtained result, making it available to the `ckpt_cond_abort()` function. If the condition $\frac{\texttt{completed_transfers}}{\texttt{total_transfers}} < \texttt{threshold}$ is verified, the function `ckpt_cond_abort()` aborts the checkpoint operation by setting a flag located in the LANai internal memory, namely `ckpt_abort`, indicating to the control program run by the LANai processor that no additional EBUS DMA data transfer associated with checkpointing needs to be programmed for that operation. If the previous condition is not verified, `ckpt_cond_abort()` spinlocks around a flag, namely `ckpt_complete`, also located in the LANai internal memory, thus resulting in temporary freezing of any simulation operation carried out by the CPU. The flag is reset by the control program as soon as the checkpoint operation gets completed.

3. Minimum cost re-synchronization

In this section we present the checkpointing-recovery cost model for non-blocking checkpointing, and the $\mathcal{MC}$ re-synchronization semantic relying on the model. Then we discuss how to solve the model and provide details on the implementation of the re-synchronization function we have developed to support the $\mathcal{MC}$ semantic within CCL.

3.1. The cost model and the $\mathcal{MC}$ re-synchronization semantic

We associate with each state vector value X traversed by the LP a probability value, namely $P(X)$, which is the probability that a future rollback needs restoration exactly to the state vector value X. Suppose we are currently taking a snapshot of the state vector value X using the non-blocking execution mode of the checkpointing protocol (i.e., the EBUS DMA is transferring data from the LP state vector into the checkpoint stack) and re-synchronization occurs either because the LP has been re-scheduled for the execution of its next event, or because some other LP needs to issue a non-blocking checkpoint request. At this point we must establish the convenience of committing/aborting the non-blocking checkpoint operation currently involving the state vector value X. Denoting with $\Delta_{\text{ckpt}}^{\text{complete}}$ the expected residual completion latency for the checkpoint operation at re-synchronization occurrence, and with $\Delta_{\text{ckpt}}^{\text{interrupt}}$ the average latency to handle checkpoint abort, the checkpointing overhead OH_{ckpt} associated with re-synchronization can be expressed as

$$OH_{\text{ckpt}} = \begin{cases} \Delta_{\text{ckpt}}^{\text{complete}} & \textit{commit case}, \\ \Delta_{\text{ckpt}}^{\text{interrupt}} & \textit{abort case}. \end{cases} \quad (1)$$

Actually, $\Delta_{\text{ckpt}}^{\text{complete}}$ corresponds to the freezing interval of CPU activities in case a commit decision is taken for the on-going checkpoint operation. $\Delta_{\text{ckpt}}^{\text{interrupt}}$ represents instead the latency for notifying to the control program run by the LANai processor that EBUS DMA data transfers associated with the on-going checkpoint operation must be interrupted.

Taking the checkpoint of the state vector value X does not affect the recovery time to any state vector value Z preceding X. This is because the state recovery time to Z depends only on the position of the latest checkpoint preceding (or coinciding with) Z, and on the granularity of the intermediate events, if any, from that checkpoint to the state vector value Z. Hence, the recovery overhead OH_{recovery} associated with re-synchronization can be evaluated as the additional expected recovery cost due to restoration of the state vector value X in case of rollback. Such a cost varies depending on whether the on-going checkpoint operation involving X is committed or aborted. Specifically, in case the checkpoint is committed, the recovery overhead due to a rollback to X is the time to reload the checkpointed state vector value X into the LP state buffer. Otherwise, coasting-forward (i.e., re-execution of intermediate events from the last taken checkpoint) is required. Denoting with (i) $EV(X)$ the set of all the events that move the LP from the latest checkpointed state vector value preceding X, to X, (ii) Δ_e the granularity (execution time) of the event $e \in EV(X)$, and (iii) Δ_{reload} the time to reload a checkpointed state vector value into the state buffer of the LP, we can express

OH_{recovery} as

$$OH_{\text{recovery}} = \begin{cases} P(X)\Delta_{\text{reload}} & \textit{commit case}, \\ P(X)\left[\Delta_{\text{reload}} + \sum_{e \in EV(X)} \Delta_e\right] & \textit{abort case}. \end{cases} \quad (2)$$

Expression (2) states that if the checkpoint operation involving X is eventually committed, then in case of state recovery to X (this happens with probability $P(X)$) the recovery overhead consists only of the time Δ_{reload} to reload the checkpointed state vector value X into the LP state buffer. Otherwise, it consists of the time to reload the latest checkpointed state vector value preceding X plus the time to replay all the events in $EV(X)$ that is, the coasting-forward time. Summing contributions in (1) and (2) we get the following expression for the whole checkpointing-recovery overhead:

$$OH_{\text{ckpt}} + OH_{\text{recovery}} = \begin{cases} \Delta_{\text{ckpt}}^{\text{complete}} + P(X)\Delta_{\text{reload}} & \textit{commit case}, \\ \Delta_{\text{ckpt}}^{\text{interrupt}} + P(X)\left[\Delta_{\text{reload}} + \sum_{e \in EV(X)} \Delta_e\right] & \textit{abort case}. \end{cases} \quad (3)$$

We note that the expression for OH_{recovery} has been derived by implicitly assuming that $P(X)$ does not change depending on whether the on-going checkpoint operation involving the state vector value X is committed or aborted. In other words, we have assumed that committing or aborting the checkpoint will result in no perceptible change in the rollback behavior. With respect to this point, we note that committing or aborting a checkpoint at a specific re-synchronization point will ultimately result in shorter or longer average distance between checkpoints for the LP, which, as commonly assumed in models underlying checkpointing techniques based on blocking (CPU charged) checkpoints [19,24,25], has in practice no significant effects on the rollback behavior.

The $\mathcal{MC}$ semantic for re-synchronization should commit/abort the on-going checkpoint operation in order to minimize the checkpointing-recovery overhead as expressed in (3). However, we note that in case the last activated non-blocking checkpoint operation is already completed at re-synchronization occurrence, no abort decision can be taken. This is reflected by the checkpointing-recovery cost model since already completed checkpoint actually means a null value for $\Delta_{\text{ckpt}}^{\text{complete}}$, which, in its turn, leads the commit decision to always minimize the checkpointing-recovery overhead. This allows us to formally define $\mathcal{MC}$ in a way that sometimes we can avoid to evaluate the cost model in (3), i.e., when the non-blocking checkpoint operation has been already completed. Denoting with x and y the values obtained by evaluating the cost model in (3) in case of commit and abort, respectively, our definition of $\mathcal{MC}$ is reported in Fig. 1. From a practical view point, the test in line 1 and the statement in line 2 allow a reduction of the overhead associated with $\mathcal{MC}$ since, in case the test is verified and the statement is executed, no calculation of $value = x - y$ must be performed (i.e., the cost model does not need to be solved upon re-synchronization).

3.2. Solving the cost model

Solving the checkpointing-recovery cost model, i.e., computing the values x and y in line 3 of the algorithm in Fig. 1, requires knowledge of several parameter values. Some of these parameters appear in the literature in performance models for classical blocking checkpointing, and a set of solutions have been already proposed for determining their values. We refer these parameters to as classical and we briefly recall in this section some of those solutions for the sake of the reader's convenience. Then we discuss how to determine the value of non-classical parameters, which are proper of the non-blocking execution mode supported by CCL.

3.2.1. Determining classical parameter values

Classical parameters appearing in performance models for blocking checkpointing [15,16,19,24,25] are (i) the event execution time, namely Δ_e, (ii) the time Δ_{reload} to reload a checkpointed state vector value into the LP state buffer, and also (iii) the probability of recovery to a given state vector value, namely $P(X)$. The first parameter has been traditionally approximated with an expected event granularity value computed over the set of granularity values characterizing the specific simulation application [19,24]. The computation is performed either off-line or during the early phase of the simulation application execution. However, for applications with large variance in the event granularity, this approach might be inadequate since the expected value might not be a reliable indicator for the granularity of a given event. For such simulation applications there exist recent solutions that point out how to measure on-line the granularity of each executed event with negligible overhead by using either fast access real time clocks or accounting instructions embedded within the application code [16]. These approaches could be straightforwardly used for keeping track of the granularity Δ_e of each already executed event e in the set $EV(X)$ characterizing our checkpointing-recovery cost model in expression (3). For what concerns Δ_{reload}, we note that computation of $value = (x - y) = (\Delta_{\text{ckpt}}^{\text{complete}} - \Delta_{\text{ckpt}}^{\text{interrupt}} - P(X) \sum_{e \in EV(X)} \Delta_e)$ in line 3 of the algorithm in Fig. 1 actually does not really need knowledge of this parameter. With respect to $P(X)$, solutions to estimate this parameter have been discussed in [4,16]. They associate with any state vector value X a left open interval of simulation time $I(X)$ evaluated as the difference between the timestamp of the

1. **if** last activated checkpoint operation already completed
2. <return COMMIT>
3. <compute $value = x - y$>
4. **if** $value \leq 0$
5. <wait for checkpoint completion>
6. <return COMMIT>
7. **else**
8. <interrupt the on-going checkpoint operation>
9. <return ABORT>

Fig. 1. The Algorithm for $\mathcal{MC}$ Re-synchronization.

event, say e', that moves the LP from the state vector value X to a successive value, and the timestamp of the event, say e, that moved the LP state vector just to the value X. More precisely, such an interval is identified as $I(X) = (timestamp(e), timestamp(e')]$. Since $P(X)$ corresponds to the probability that a timestamp order violation will eventually occur in the interval $I(X)$, [3] these solutions estimate $P(X)$ exploiting the length of the interval $I(X)$ and monitoring the relative frequency of timestamp order violation occurrences in simulation time intervals of a given length. In other words, these solutions maintain a histogram for the relative frequency of state recovery in simulation time intervals of pre-specified lengths. How to determine the length of those intervals in order to get balanced observations has been also discussed in [4]. We note that if the histogram maintains data related to a unique left open interval $[0, \infty)$, then $P(X)$ is simply estimated as the *rollback frequency* of the LP, namely the ratio between the total number of rollbacks and the total number of event executions at the LP.

3.2.2. Determining non-classical parameter values

Two parameters appear in the cost model in (3), whose value determination has not been already treated in the literature, since they do not appear in any already proposed performance model for blocking checkpointing. They are the expected residual completion latency of the checkpoint operation $\Delta_{ckpt}^{complete}$ and the time to handle checkpoint abort $\Delta_{ckpt}^{interrupt}$. We discuss in this section how to deal with these parameters in the specific CCL implementation.

3.2.2.1. Determining $\Delta_{ckpt}^{interrupt}$: As already discussed in [17] while presenting the $\mathcal{CCA}$ semantic for re-synchronization, checkpoint abort only requires notification of the abort decision to the LANai control program. This has been implemented by letting the function `ckpt_cond_abort()` (run at the host side) set a flag, namely `ckpt_abort`, implemented as a word located into the LANai internal memory. Flag setting indicates to the LANai control program that any on-going checkpoint operation must be interrupted. Actually, software run at the host side must execute no other action for handling checkpoint abort since all the work for aborting the checkpoint operation (i.e., for interrupting data transfer associated with checkpointing) is carried out by the control program. We maintain this approach also for the implementation of the $\mathcal{MC}$ semantic, therefore the only overhead experienced at the host side for handling checkpoint abort is the latency to set the flag value into the LANai internal memory. $\Delta_{ckpt}^{interrupt}$ corresponds exactly to this latency value, evaluated in case the last activated non-blocking checkpoint operation has not yet been completed. With respect to the latter assertion, we recall that, by the definition of $\mathcal{MC}$ in Fig. 1, the cost model must be solved only in case the last activated non-blocking checkpoint operation is still in progress, therefore we need the value of $\Delta_{ckpt}^{interrupt}$ just in case the EBUS DMA is active either for data transfer associated with checkpointing or for higher priority transfers of messages incoming from the network into the receive queue.

The host CPU and the EBUS DMA have the same access priority to the LBUS (see the specifications in [11,12]). Also, other components on board of the card (e.g., the LANai processor and the other DMA engines) have lower access priority to the LBUS (see again [11,12]). As a consequence, given that the flag is set using a single bus cycle, we expect minimal variance for the latency to set the flag value despite the fact that the EBUS DMA is working either for data transfer associated with checkpointing or for message transfer into the receive queue. Specifically data transfer currently performed by the EBUS DMA can delay the flag update performed by the host CPU of at most a single bus cycle. As an empirical support to previous deduction, we have measured the time to set the flag value in case the EBUS DMA continuously performs data transfer operations, from/to the host memory, involving data blocks of different sizes ranging from 4 bytes to 8 Kbytes. The sequence of activations of the EBUS DMA is such that a data block is first copied from the host memory into the LANai internal memory and then is copied back from the LANai internal memory into the host memory. The measures have been taken for a M2M-PCI32C myrinet card, mounted on a Pentium II 300 MHz (the ex-

[3] After e' is executed, the violation in the interval $I(X)$ can be caused by the scheduling for that LP of an event e'' with $timestamp(e'') \in I(X)$, or even by the arrival of the antimessage that cancels e'.

Table 1
Flag setting latency vs data block size (average over 1000 samples)

Data block size (bytes)	4	16	512	1024	2048	4096	8192
Flag setting latency (microseconds)	0.84	0.84	0.84	0.84	0.83	0.84	0.84

perimental data presented in Section 4 have been taken on a cluster of 8 of these machines). The results reported in Table 1 show that the latency to set the flag value is independent of the data block size being transferred by the EBUS DMA. Therefore, such a value can be used as a reliable measure for $\Delta_{ckpt}^{interrupt}$ while solving the checkpointing-recovery cost model. From a pragmatical view point, this latency value can be measured once upon the installation of CCL on the specific hardware/software architecture. The measurement can be performed through adequate benchmarks, such as the one we have used for the previously described experiments. Then the function implementing $\mathcal{MC}$ can use that measured value to solve the cost model.

3.2.2.2. Determining $\Delta_{ckpt}^{complete}$*:* By the description in Section 2, the counters `total_transfers` and `completed_transfers` indicate the total amount of bursts required by the last activated non-blocking checkpoint operation and the amount of those bursts which have already been completed. Hence, the difference between the two counter values indicates the amount of bursts that still need to be carried out in order to complete the checkpoint operation. Denoting with Δ_{burst} the time for a single burst performed by the EBUS DMA, we get the following lower bound L for $\Delta_{ckpt}^{complete}$:

$$L = (total\ transfers - completed\ transfers)\Delta_{burst}. \quad (4)$$

This lower bound is obtained in case no message needs to be transferred into the receive queue through the EBUS DMA during a period that starts just upon re-synchronization occurrence and ends with the completion of the checkpoint operation. We refer to this period as *re-synchronization period*. However, it is possible that messages incoming from the network need to be transferred into the receive queue through the EBUS DMA during the re-synchronization period. Given that message transfer has higher priority as compared to data transfer associated with checkpointing, the real value of $\Delta_{ckpt}^{complete}$ might be actually larger that the lower bound L. The real value of $\Delta_{ckpt}^{complete}$ actually corresponds to the re-synchronization period length, therefore, our objective is to evaluate the expected length of the re-synchronization period. Denoting with M the number of messages already buffered into the LANai internal memory upon re-synchronization occurrence, which need to be transferred into the receive queue through the EBUS DMA, and with $\Delta_{message}$ the time required by the EBUS DMA for transferring a single message from the LANai internal memory into the receive queue, the length of the re-synchronization period $\Delta_{ckpt}^{complete}$ can be expressed as

$$\Delta_{ckpt}^{complete} = L + M\Delta_{message} + L', \quad (5)$$

where L is the previously mentioned lower bound, $M\Delta_{message}$ is the time for transferring those M messages into the receive queue through the EBUS DMA, and L' is the time required for transferring into the receive queue the additional messages incoming from the network during the re-synchronization period. Denoting with f the expected frequency of message arrival from the network during the re-synchronization period we get

$$L' = f\Delta_{ckpt}^{complete}\Delta_{message}, \quad (6)$$

where $f\Delta_{ckpt}^{complete}$ represents the amount of messages incoming from the network during the re-synchronization period. Plugging (4) and (6) in (5), we can express $\Delta_{ckpt}^{complete}$ as [4]

$$\Delta_{ckpt}^{complete} = \begin{cases} \frac{(total\ transfers - completed\ transfers)\Delta_{burst} + M\Delta_{message}}{1 - f\Delta_{message}} & if\ f\Delta_{message} < 1, \\ \infty & if\ f\Delta_{message} \geqslant 1. \end{cases} \quad (7)$$

By (7), calculation of $\Delta_{ckpt}^{complete}$ requires knowledge of M, Δ_{burst}, $\Delta_{message}$ and f. The current implementation of CCL already maintains a counter in the LANai internal memory (managed by the LANai control program and accessible by software run at the host side) which keeps track at any instant of the amount of messages already buffered into the LANai internal memory, which need to be transferred into the receive queue. Therefore, the value of M is straightforwardly available to the re-synchronization function implementing $\mathcal{MC}$ just through this counter value.

In case a commit decision is taken, the re-synchronization function must wait for the completion of the on-going checkpoint operation in a way to temporarily suspend any other simulation operation carried out by the CPU. As for the case of $\mathcal{CCA}$, the re-synchronization function we have developed to implement $\mathcal{MC}$ spinlocks around the flag `ckpt_complete`, which is set to one by the control program as soon as the operation is completed. As a consequence, the values of Δ_{burst} and $\Delta_{message}$ must be evaluated

[4] $\Delta_{ckpt}^{complete}$ has been set equal to infinity also in the case of $f\Delta_{message}>1$ since a strict application of algebra would lead to the physical absurd of a negative value for $\Delta_{ckpt}^{complete}$ anytime $f\Delta_{message} > 1$.

considering the interference on EBUS DMA activities due to access of the host CPU to the PCI bridge and to the LBUS caused by read operations continuously issued on the flag `ckpt_complete` while waiting for the completion of the checkpoint operation. (As already pointed out, all the other components on board of the card have lower access priority to the LBUS as compared to the EBUS DMA [13], therefore they do not produce interference on data transfer performed by the EBUS DMA.) With respect to this point, measures reported in [18] have shown that the time to transfer a data block from/to the host memory through the EBUS DMA while the host CPU spinlocks around a flag located in the LANai internal memory increases linearly with the data block size. As a consequence, once fixed the message size and the burst size, Δ_{burst} and Δ_{message} can be assumed as constant values any time we hypothesize that no other PCI peripheral (such as an audio card) is using the same PCI bus. Since this can be considered as a common scenario when dedicating a cluster to a specific parallel computing application, Δ_{burst} and Δ_{message} can be determined while installing CCL (for example through the benchmarking described in [18]) and made available to the re-synchronization function implementing $\mathcal{MC}$.

By the previous arguments, f is the only parameter that depends on proper dynamics of the specific parallel execution, whose value really needs to be determined upon re-synchronization occurrence. As pointed out before, f is the expected frequency of message arrival during the re-synchronization period, however we need this value at the start of the period itself in order to solve the cost model, therefore a prediction mechanism for determining this value must be encompassed. According to the commonly accepted belief that the recent past behavior should be a reliable indicator of the immediate future behavior (for instance this is the basic assumption underlying many dispatcher designs, where recent process/thread behavior, in terms of CPU bursts or I/O requests is assumed as representative of the immediate future behavior [23,26]), we have decided to approximate f with the frequency of message arrival from the network in the interval between the activation of the last non-blocking checkpoint operation and the occurrence of re-synchronization. To compute that frequency value, we exploit hardware features of the LANai chip. Specifically, this chip is equipped with a Real Time Clock register (RTC), providing real time measures with granularity on the order of 0.5 microseconds, which can be reset/read by the host CPU. We have introduced an additional counter in the LANai internal memory, namely `incoming_messages`, which counts the amount of messages incoming from the network since the activation of the last non-blocking checkpoint operation. The counter is incremented by the control program each time it activates the Receive DMA on board of the card. Then we have slightly modified the function `non_block_ckpt()` in order to let it reset both `incoming_messages` and the register RTC upon the issue of a non-blocking checkpoint operation. The value of f to be used while computing $\Delta_{\text{ckpt}}^{\text{complete}}$ can be evaluated by the re-synchronization function implementing $\mathcal{MC}$ as

$$f = \frac{\textit{incoming messages}}{\$RTC}. \tag{8}$$

3.3. Details on the re-synchronization function implementing $\mathcal{MC}$

The function implementing $\mathcal{MC}$ has the prototype `min_cost_resynch(double prob, double cumulate, int option)`, where `prob` corresponds to the probability value $P(X)$, `cumulate` is the sum of the execution costs of all the events executed by the LP since the last committed checkpoint operation, expressed in microseconds, and `option` is a value that can be selected among macros in the set {`DEFAULT`, `ALWAYS_COMMIT`, `ALWAYS_ABORT`}, which determines one of the following three run time behaviors: `DEFAULT` - the `min_cost_resynch()` function commits/aborts the checkpoint on the basis of the $\mathcal{MC}$ semantic, exactly as defined by the algorithm in Fig. 1. `ALWAYS_COMMIT` - the `min_cost_resynch()` function always waits for the completion of any on-going checkpoint operation. This is done by simply spinlocking around the flag `ckpt_complete`. `ALWAYS_ABORT` - the `min_cost_resynch()` function always aborts an on-going checkpoint operation. This is done by simply setting the flag `ckpt_abort` in the LANai internal memory. The `DEFAULT` value for `option` is straightforward and does not deserve further discussion. The reasons why we have introduced the other two options are instead discussed in the following paragraphs.

`ALWAYS_COMMIT` **option**: If checkpoint commit is determined only on the basis of the $\mathcal{MC}$ semantic as described by the algorithm in Fig. 1, then there is the possibility that very few checkpoints are committed for a given LP. (This might happen, for example, when the parameter $P(X)$ in the cost model approaches the value zero, and upon re-synchronization occurrence, we frequently find that the last activated checkpoint operation is still in progress.) As widely known, this might originate a negative interference with the frequency of Global Virtual Time (GVT) calculation and fossil collection [15]. To prevent this problem, we can force commitment of an on-going checkpoint operation for an LP in case the number of executed events from the last committed checkpoint of that LP oversteps a given threshold. This can be done just exploiting the `ALWAYS_COMMIT` option. How to select that threshold value has been discussed in several works dealing with blocking checkpointing [5,19].

`ALWAYS_ABORT` **option**: In case the scheduled LP must roll back, and a checkpoint operation of its own state vector is still in progress upon re-synchronization, it is better to abort the operation. This is because the current value of the state vector of the LP, which is being checkpointed, represents incorrect information to be discarded (i.e., overwrit-

ten) during the state recovery procedure. To ensure the abort of that useless checkpoint, the `ALWAYS_ABORT` option can be used.

4. Experimental analysis

In [20], results of an extended performance study of non-blocking checkpointing with $\mathcal{MC}$ are shown. For space limitations we cannot report that complete set of data, therefore we present data related to optimistic parallel simulation of a specific Personal Communication System (PCS) model, selected as a representative real world application among those employed in [20]. Before reporting the performance data, we provide details on the test settings, the simulation model and the metrics we have selected in the experimental study.

4.1. Test settings

We have used a cluster of 8 Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache) running LINUX (kernel version 2.0.32) and equipped with M2M-PCI32C myrinet cards. The simulation software implements the events as a compound structure with several fields (receiver, timestamp etc.), having total size of 60 bytes. Using CCL any message carrying an event is delivered to the recipient within about 20 ms when no congestion occurs on the myrinet switch. LPs are implemented as application-level threads and there is an instance of an optimistic simulation engine on each machine. Message exchange among LPs hosted by the same machine does not involve operations of the CCL layer. The engine manages the local event list (resulting as the logical collection of the event lists of the local LPs) and schedules LPs for event execution according to the standard Smallest-Timestamp-First algorithm [8]. Dynamic memory allocation/release based on standard `malloc()/free()` calls is adopted for the entries of the event lists.

We do not limit our analysis to a pure comparison between the two re-synchronization semantics $\mathcal{MC}$ and $\mathcal{CCA}$ in support of non-blocking checkpointing, since we also report results related to blocking (CPU charged) checkpointing in combination with a classical Periodic State Saving (PSS) approach that makes the LP to take a checkpoint each χ event executions [5,15,19] (recall χ is commonly referred to as checkpoint interval).[5] Blocking checkpointing is implemented efficiently through a `memcpy()` call. Furthermore, as CCL adopts reserved main memory pages for both the LPs state buffers and their checkpoint stacks, to ensure fairness in the comparison we have used reserved main memory pages also for blocking checkpointing.

For the tests with non-blocking checkpointing, 1 Kbyte has been selected as the burst length for EBUS DMA operations on the basis of measurements reported in [18]. Also, a non-blocking checkpoint operation is activated by the LP after the execution of each simulation event, thus obtaining a situation in which lack of checkpoints for particular LP state vector values is determined only on the basis of checkpoint abort decisions taken upon re-synchronization. As shown in [17], this is a favorable test case for non-blocking checkpointing, allowing full exploitation of the effectiveness of an optimized re-synchronization semantic. We have imposed a maximum distance of 20 events between two successive committed checkpoints of any LP in order not to incur problems related to the interaction between checkpointing and the frequency of GVT calculation (see Section 3.3).

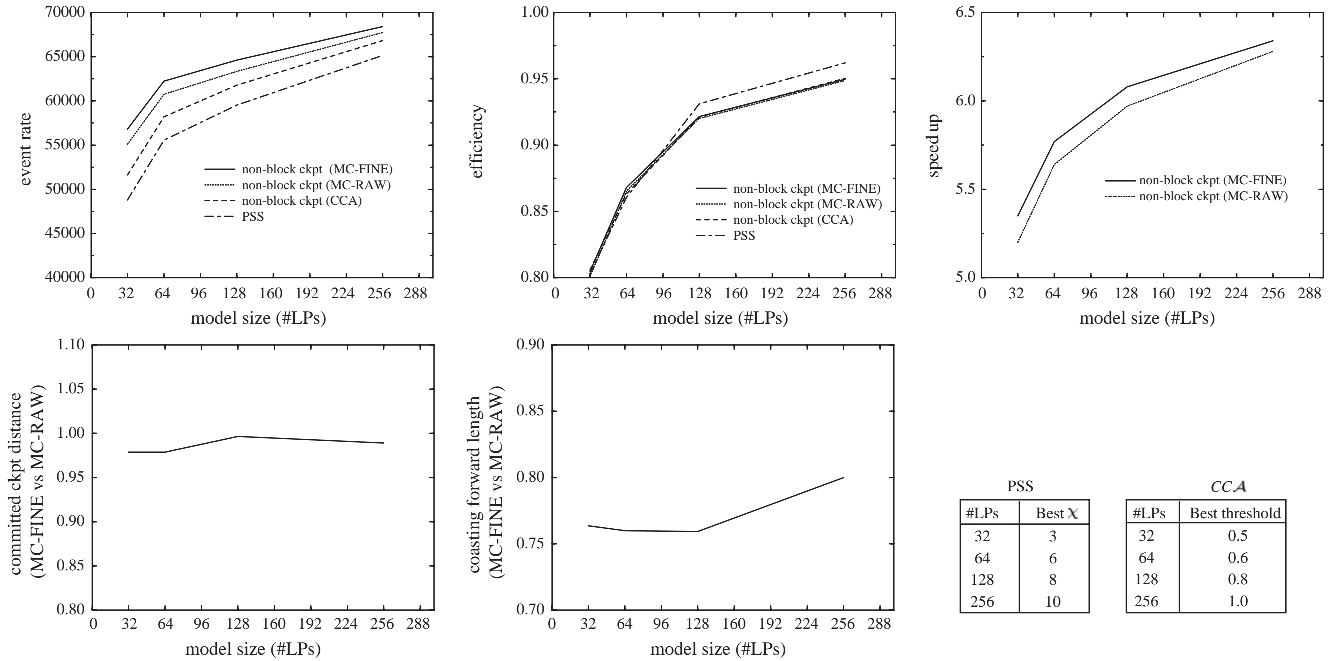
As a last preliminary observation, the expected event granularity for the used PCS model is known in advance, therefore we have used such an expected value as the execution cost Δ_e of any event in the set $EV(X)$ of the cost model (see expression (3)). For what concerns the parameter $P(X)$, namely the probability of rollback to a given state vector X, we have adopted the following two different estimation approaches to support $\mathcal{MC}$: *Raw Estimation*—$P(X)$ is simply estimated as the rollback frequency of the LP (i.e., the ratio between the number of rollbacks and the number of executed events at that LP). *Fine Estimation*—we maintain a histogram for the relative frequency of state recovery in simulation time intervals of a pre-specified length.[6] Then, as discussed in Section 3.2.1, $P(X)$ is evaluated as the relative frequency associated with the interval that contains $I(X)$.

4.2. Simulation model

In our PCS simulation model the service area is partitioned into cells, each of which is modeled by a distinct LP. Each cell represents a receiver/transmitter having a fixed number of 100 channels allocated to it. The model is call-initiated [3] since it only simulates the behavior of a mobile unit during conversation. Call requests arrive to each cell according to an exponential distribution [1–3] with inter-arrival time 1.5 s, and the average call holding time is 2 min. There are two distinct classes of mobile units, both of them characterized by a residence time within a cell which follows an exponential distribution, with mean 3 min (fast movement units) and 30 min (slow movement units), respectively. Also, any call is equally likely to be destined to a fast or a slow movement mobile unit. The state vector of any LP records statistics, information about busy channels and, for each channel, information about features of the on-going call (e.g. scheduled call termination time, class of the mobile unit, etc.), if any. As a result, the size of the state vector

[5] Although for some particular simulation problems (e.g., simulations with very regular patterns in the difference between timestamps of consecutive events) performance improvements over PSS can be achieved by relaxing the constraint that CPU charged checkpoints must be taken on a periodic basis [16], PSS remains, in general, a performance effective solution. Hence, we take it as a reference point in the experimental study.

[6] The length has been stepped by 0.1 times the average simulation time increment at an LP.



PSS

#LPs	Best χ
32	3
64	6
128	8
256	10

$\mathcal{CCA}$

#LPs	Best threshold
32	0.5
64	0.6
128	0.8
256	1.0

Fig. 2. Results for the PCS simulation application.

is about 4 Kbytes. The expected event granularity for this model, including the cost of statistics update, is on the order of 35 ms on the adopted architecture. We have simulated a PCS with hexagonal cells. The model size, in terms of number of cells, i.e., number of LPs, has been varied between 32 and 256. The LPs have been evenly distributed among the 8 machines of the cluster.

4.3. Metrics

We report results related to the *event rate*, classically evaluated as the number of committed simulation events per second. This parameter indicates how fast is the parallel execution with a given checkpointing/re-synchronization scheme. For the case of non-blocking checkpointing with $\mathcal{CCA}$, we report the peak event rate observed while moving the parameter `threshold` from 0.0 to 1.0, with step 0.1. Similarly, for PSS we report the peak event rate observed vs χ. The checkpoint interval χ and the `threshold` value originating those peak event rate values are also reported. We additionally report results related to the *efficiency* of the parallel execution, classically evaluated as the ratio between the amount of committed simulation events and the total number of executed events (committed plus rolled back). This parameter is an indicator of the percentage of productive simulation work performed since it is a measure of the probability for an event not to be eventually rolled back once executed. For $\mathcal{MC}$ we also report the speedup achieved by the parallel execution, namely the ratio between serial execution time of the same simulation model on a single machine and parallel execution time on the 8 machines. This parameter allows us to ascertain whether performance results are obtained with an effective parallel implementation. Each reported value is the average of 20 runs, all done with different seeds for the pseudo-random generators. At least 5×10^6 committed simulation events have been executed in each run.

4.4. Results

The plots for the event rate in Fig. 2 show a clear gain in the execution speed obtained by non-blocking checkpointing through $\mathcal{MC}$. For the case of $\mathcal{MC}$-FINE, such a gain is up to 10% as compared to the event rate achieved with $\mathcal{CCA}$ and, more important, it is up to 16% as compared to the event rate of PSS. Slightly reduced gains are achieved for the case of $\mathcal{MC}$-RAW, however they still remain on the order of 7% compared to $\mathcal{CCA}$ and 13% compared to PSS. In general, higher gains in the execution speed are observed in case of smaller size of the simulation model, which originates a higher degree of parallelism in the model execution. This is a natural behavior that can be easily explained when looking at the efficiency plots. Specifically, when the degree of parallelism in the model execution gets lower, i.e., the model size is increased, we obtain efficiency values on the order of up to 95%, denoting a very low amount of rollback in the parallel execution (events are less likely to be rolled back once executed). In this situation, any optimized checkpointing approach, whether it be based on blocking or non-blocking checkpointing, typically allows us to almost eliminate the checkpointing overhead while paying in practice negligible coasting-forward costs due to the fact that rollbacks are infrequent. With respect to this point, in case of model size

256, PSS achieves the best performance for a large value of the checkpoint interval χ, which just indicates the strong reduction of the CPU overhead due to checkpointing, even in case of blocking checkpoints, at the point in which the event rate is maximized vs χ. A similar observation can be drawn for non-blocking checkpointing with $\mathcal{CCA}$ when considering that, for model size 256, the best `threshold` value is 1.0, which means that any on-going checkpoint is actually aborted upon re-synchronization. On the other hand, when the amount of rollback is non-minimal, i.e., for efficiency values on the order of 85% or less in our experiments, non-blocking checkpointing with $\mathcal{MC}$ allows significant performance improvements. The speedup achieved by non-blocking checkpointing ranges between 65% and 78% of the ideal speedup on 8 machines for the case of $\mathcal{MC}$-RAW, while it ranges between 67% and 79% of the ideal one for the case of $\mathcal{MC}$-FINE. This is an indication that performance data reported in this study are representative, since they are related to an efficient parallel implementation.

We argue that the gain of $\mathcal{MC}$-FINE over $\mathcal{MC}$-RAW derives from the fact that the fine estimation method for $P(X)$ better identifies those state vectors that are more likely to be eventually recovered. Therefore, compared to $\mathcal{MC}$-RAW, $\mathcal{MC}$-FINE should take more commit decisions for checkpoints associated with those state vectors. Given that simulation events for this specific model have the same expected granularity, the effects of such a better placement of committed checkpoints should allow a reduction of the average coasting-forward length. To support this deduction we report in Fig. 2 (i) the ratio between the average distance of successive committed checkpoints of both $\mathcal{MC}$-FINE and $\mathcal{MC}$-RAW, and also (ii) the ratio between the average length of coasting-forward achieved by the two schemes. By the plots, $\mathcal{MC}$-FINE and $\mathcal{MC}$-RAW exhibit in practice the same distance between committed checkpoints, which indicates that the checkpointing overhead paid by the two schemes is similar. On the other hand, $\mathcal{MC}$-FINE exhibits shorter coasting-forward (up to 20/25%), which allows a reduction of the recovery cost. This reduction lets $\mathcal{MC}$-FINE gain over $\mathcal{MC}$-RAW especially for a larger amount of rollback in the parallel execution, i.e., for a larger degree of parallelism in the model execution. This is because, as discussed above, the achievement of a better tradeoff between checkpointing and recovery cost is a major issue, and really helps performance, just when the impact of rollback is larger.

References

[1] A. Boukerche, S.K. Das, A. Fabbri, O. Yildz, Exploiting model independence for parallel PCS network simulation, in: Proceedings of the 13th Workshop on Parallel and Distributed Simulation, ACM/IEEE Computer Society, May 1999, pp. 166–173.

[2] C.D. Carothers, D. Bauer, S. Pearce, ROSS: a high performance modular Time Warp system, in: Proceedings of the 14th Workshop on Parallel and Distributed Simulation, ACM/IEEE Computer Society, May 2000, pp. 53–60.

[3] C.D. Carothers, R.M. Fujimoto, Y.B. Lin, A case study in simulating PCS networks using Time Warp, in: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, ACM/SCS, June 1995, pp. 87–94.

[4] A. Ferscha, J. Luthi, Estimating rollback overhead for optimism control in Time Warp, in: Proceedings of the 28th Annual Simulation Symposium, IEEE Computer Society, April 1995, pp. 2–12.

[5] J. Fleischmann, P. Wilsey, Comparative analysis of periodic state saving techniques in Time Warp simulators, in: Proceedings of the 9th Workshop on Parallel and Distributed Simulation, ACM/SCS, June 1995, pp. 50–58.

[6] R.M. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (10) (October 1990) 30–53.

[7] D.R. Jefferson, Virtual time, ACM Trans. Programm. Languages System 7 (3) (July 1985) 404–425.

[8] Y.B. Lin, E.D. Lazowska, Processor scheduling for Time Warp parallel simulation, Advances in Parallel and Distributed Simulation, 1991, pp. 11–14.

[9] MYRICOM, http://www.myri.com/news/03623/index.html.

[10] MYRICOM, http://www.myri.com, 1994.

[11] MYRICOM, LANai 4. Draft, February 1999.

[12] MYRICOM, LANai 9, June 2000.

[13] MYRICOM, PCI64 programmer's documentation, May 2001.

[14] S. Pakin, M. Lauria, A. Chen, High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet, in: Proceedings of Supercomputing'95, ACM/IEEE Computer Society, December 1995.

[15] B.R. Preiss, W.M. Loucks, D. MacIntyre, Effects of the checkpoint interval on time and space in Time Warp, ACM Trans. Model. Comput. Simul. 4 (3) (July 1994) 223–253.

[16] F. Quaglia, A cost model for selecting checkpoint positions in Time Warp parallel simulation, IEEE Trans. Parallel Distrib. Systems 12 (4) (February 2001) 346–362.

[17] F. Quaglia, A. Santoro, Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation, IEEE Trans. Parallel Distrib. Systems 14 (6) (June 2003) 593–610.

[18] F. Quaglia, A. Santoro, B. Ciciani, Tuning of the checkpointing and communication library for optimistic simulation on Myrinet based NOWs, in: Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE Computer Society, Silver Spring, MD, October 2001, pp. 241–248.

[19] R. Ronngren, R. Ayani, Adaptive checkpointing in Time Warp, in: Proceedings of the 8th Workshop on Parallel and Distributed Simulation, ACM/SCS, July 1994, pp. 110–117.

[20] A. Santoro, Semi-asynchronous checkpointing for optimistic parallel simulation, Ph.D. Thesis, Dipartimento di Informatica e Sistemistica, University of Rome La Sapienza, February 2003.

[21] A. Santoro, F. Quaglia, Benefits from semi-asynchronous checkpointing for Time Warp simulations of a large state PCS model, in: Proceedings of the Winter Simulation Conference, Society for Computer Simulation, December 2001, pp. 1339–1345.

[22] A. Santoro, F. Quaglia, Checkpointing and Communication Library (CCL), http://www.dis.uniroma1.it/PADS/software/CCL, 2003.

[23] A. Silberschatz, P. Galvin, Operating System Concepts, Addison-Wesley Publishing Company, Reading, MA, 1994.

[24] S. Skold, R. Ronngren, Event sensitive state saving in Time Warp parallel discrete event simulation, in: Proceedings of the Winter Simulation Conference, Society for Computer Simulation, December 1996, pp. 653–660.

[25] H. Soliman, A. Elmaghraby, An analytical model for hybrid checkpointing in Time Warp distributed simulation, IEEE Trans. Parallel Distrib. Systems 9 (10) (October 1998) 947–951.

[26] W. Stallings, Operating Systems Internals and Design Principles, Prentice-Hall, Englewood Cliffs, NJ, 1998.

Francesco Quaglia received the Laurea degree (MS level) in Electronic Engineering in 1995 and the Ph.D. degree in Computer Engineering in 1999 from the University of Rome "La Sapienza". From summer 1999 to summer 2000 he held an appointment as a Researcher at the Italian National Research Council (CNR). Since January 2005 he works as an Associate Professor at the School of Engineering of the University of Rome "La Sapienza", where he has previously worked as an Assistant Professor since September 2000 to December 2004. His research interests include parallel discrete event simulation, parallel computing, distributed systems, fault-tolerant programming and performance evaluation of software/hardware systems. He regularly serves as a referee for several international conferences and journals. He serves on the program committees of the *Workshop on Principles of Advanced and Distributed Simulation* (PADS) and of the *Symposium on Distributed Simulation and Real Time Applications* (DS-RT), and is a member of the editorial board of the *International Journal of Simulation and Process Modelling*. He has also served as Program Co-Chair of the 16th edition of PADS.

Andrea Santoro received the "Laurea" degree (MS level) in Electronic Engineering from the University of Rome "La Sapienza", and the Ph.D. in Computer Engineering from the same institution. He worked at Georgia Tech in 2003 and is currently working as a post-doc at the University of Rome "La Sapienza". He is an IEEE Member. His research interests are in the fields of parallel simulation, operating systems and networks. He serves as a referee for several international journals and conferences in the same areas, where he also published several papers.